Programowanie obiektowe w PHP 5

Autor: Hasin Hayder
ISBN: 978-83-246-1821-7
Tytuł oryginału: Object-Oriented Programming with PHP5
Format: 170x230, stron: 264

- Naucz się definiować właściwości obiektów
- Stwórz kod, który będzie łatwy w zarządzaniu
- Zbuduj wydajną i bezpieczną aplikację

Programowanie obiektowe (OOP) wciąż zyskuje rzesze nowych zwolenników. Ponieważ opiera się ono na klasach i obiektach, jest znacznie bardziej intuicyjne niż programowanie strukturalne. Do jego podstawowych zalet zaliczyć należy także łatwość modyfikowania oraz możliwość wielokrotnego wykorzystania klas. PHP 5 udostępnia wiele różnorodnych mechanizmów (na przykład obsługę wyjątków czy zbiór interfejsów znacznie rozszerzających możliwości klas użytkownika) oraz pełny moduł obsługujący styl programowania OOP, dzięki czemu jest doskonałym narzędziem, pozwalającym tworzyć wydajne, bezpieczne i dynamiczne aplikacje z wykorzystaniem programowania obiektowego.

Książka „Programowanie obiektowe w PHP 5" jest doskonałym źródłem informacji, które pomoże Ci zrozumieć najistotniejsze koncepcje programowania zorientowanego obiektowo w PHP 5. Podręcznik zawiera omówienie zagadnień podstawowych oraz bardziej zaawansowanych, takich jak architektura Model-View-Controller (MVC) oraz testy jednostkowe. Znajdziesz tu także praktyczne wskazówki i przykłady dotyczące m.in. użycia biblioteki Standard PHP Library. Dowiesz się, jak używać odpowiedniego wzorca, aby zwiększyć wydajność kodu, czym jest testowanie jednostkowe i dlaczego stanowi ono zasadniczą część tworzenia dobrego oraz stabilnego projektu programu. Nauczysz się tworzyć wydajne, bezpieczne i łatwe w zarządzaniu aplikacje.

- Praca z OOP – tworzenie obiektów
- Funkcje dostarczające informacje o klasie
- Iteratory
- Automatyczne wczytywanie klas
- Serializacja
- Wzorce projektowe
- Refleksja i testy jednostkowe
- Biblioteka Standard PHP Library
- Obsługa baz danych z użyciem stylu OOP
- Używanie architektury MVC

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 2

Rozpoczęcie pracy z OOP

W tym rozdziale czytelnik dowie się, w jaki sposób tworzyć obiekty, definiować ich atrybuty (czyli właściwości) oraz metody. W języku PHP obiekty zawsze są tworzone za pomocą słowa kluczowego `class`. Po lekturze rozdziału czytelnik znacznie rozszerzy wiedzę z zakresu klas, właściwości i metod. Ponadto w rozdziale tym zostaną podjęte tematy związane z zasięgiem metod, modyfikatorami metod oraz zostaną przedstawione zalety płynące z używania interfejsów. Niniejszy rozdział jest także wprowadzeniem do innych podstawowych funkcji programowania zorientowanego obiektowo w PHP. Biorąc to wszystko pod uwagę, można zaryzykować stwierdzenie, że ten rozdział jest jednym z lepszych zasobów pozwalających na rozpoczęcie pracy z OOP w języku PHP.

Tworzenie obiektów

Jak wcześniej wspomniano, obiekt w języku PHP jest tworzony za pomocą słowa kluczowego `class`. Wymieniona klasa składa się z właściwości i metod (publicznych bądź prywatnych). Przyjrzyjmy się klasie `Emailer`, która została już przedstawiona w rozdziale 1. Teraz przeanalizujemy sposób działania klasy `Emailer`:

```
<?
// class.emailer.php
class Emailer
{
   private $sender;
   private $recipients;
   private $subject;
   private $body;
   function __construct($sender)
   {
      $this->sender = $sender;
```

```
        $this->recipients = array();
    }
    public function addRecipients($recipient)
    {
        array_push($this->recipients, $recipient);
    }
    public function setSubject($subject)
    {
        $this->subject = $subject;
    }
    public function setBody($body)
    {
        $this->body = $body;
    }
    public function sendEmail()
    {
        foreach ($this->recipients as $recipient)
        {
            $result = mail($recipient, $this->subject, $this->body,
                "From: {$this->sender}\r\n");
            if ($result) echo "Wiadomość została wysła do
                {$recipient}<br/>";
        }
    }
}
?>
```

Powyższy kod rozpoczyna się poleceniem `class Emailer`, które oznacza, że nazwa tworzonej klasy to `Emailer`. Podczas nadawania nazw klasom należy stosować tę samą konwencję nazw, która jest używana w stosunku do zmiennych, na przykład nazwy nie rozpoczynamy od cyfry, itd.

Następnie wiersze kodu odpowiadają za deklarację właściwości klasy. Możemy więc wyodrębnić cztery — `$sender`, `$recipient`, `$subject` oraz `$body`. Warto zwrócić uwagę, że każda z wymienionych właściwości została zadeklarowana z użyciem słowa kluczowego `private` (prywatna). Właściwość prywatna to taka, która jest dostępna jedynie w danej klasie. Trzeba jeszcze dodać, że właściwości to po prostu zmienne wewnątrz klasy.

Jak czytelnik zapewne pamięta, metoda jest po prostu funkcją zdefiniowaną wewnątrz klasy. W klasie przedstawionej na powyższym kodzie znajduje się pięć funkcji — `__construct()`, `addRecipient()`, `setSubject()`, `setBody()` oraz `sendEmail()`. Warto zwrócić uwagę, że ostatnie cztery metody zostały zadeklarowane z użyciem słowa kluczowego `public` (publiczne). Oznacza to, że każdy, kto utworzy egzemplarz tego obiektu, posiada również dostęp do jego metod publicznych.

Metoda `__construct()` jest metodą specjalnego znaczenia w klasie i jest nazywana *konstruktorem*. W trakcie tworzenia nowego obiektu na podstawie klasy metoda konstruktora jest automatycznie wywoływana. Dlatego też, jeśli podczas tworzenia obiektu trzeba na nim wykonać

określone zadania, to najlepszym rozwiązaniem jest zdefiniowanie ich w konstruktorze. Przykładowo, metoda konstruktora klasy `Emailer` powoduje zdefiniowanie pustej tablicy `$recipients` oraz danych nadawcy.

Dostęp do właściwości i metod z wewnątrz klasy

Czytelnik zapewne zastanawia się, w jaki sposób funkcje mogą uzyskać dostęp do właściwości klasy z poziomu danej klasy? Do tego celu służy następująca konstrukcja kodu:

```
public function setBody($body)
{
   $this->body = $body;
}
```

W klasie znajduje się właściwość prywatna o nazwie `$body`. Jeżeli zachodzi potrzeba uzyskania dostępu do niej z wewnątrz funkcji, wtedy należy użyć słowa kluczowego `$this`. Wymienione słowo kluczowe `$this` oznacza odniesienie do bieżącego egzemplarza obiektu. Dlatego też, aby uzyskać dostęp do właściwości `body`, trzeba zastosować polecenie `$this->body`. Warto zwrócić uwagę, że w celu uzyskania dostępu do właściwości (na przykład zmiennych) klasy trzeba użyć operatora „`->`”, a następnie nazwy egzemplarza.

Podobnie jak w przypadku właściwości, także dostęp do metod klasy z poziomu innej metody klasy odbywa się z pomocą przedstawionej powyżej konstrukcji. Przykładowo, wywołanie metody `setSubject` następuje w przedstawiony sposób: `$this->setSubject()`.

> Warto zwrócić uwagę, że słowo kluczowe `$this` jest poprawne tylko w zasięgu metody, która nie została zadeklarowana jako `static` (statyczna). Słowa kluczowego `$this` nie można użyć z zewnątrz klasy. Więcej informacji na temat modyfikatorów `static`, `private` i `public` zostanie przedstawionych w podrozdziale Modyfikatory znajdującym się w dalszej części rozdziału.

Używanie obiektu

Nadeszła pora na użycie nowo utworzonego obiektu `Emailer` w kodzie PHP. W tym miejscu trzeba dodać, że przed użyciem obiektu należy wykonać pewne przygotowania. Przede wszystkim, zanim obiekt będzie mógł zostać użyty, wcześniej musi być utworzony jego egzemplarz. Po utworzeniu egzemplarza obiektu programista zyskuje dostęp do jego wszystkich publicznych właściwości i metod poprzez użycie operatora „`->`” po nazwie obiektu. W poniższym fragmencie kodu przedstawiono przykładowe użycie obiektu:

```
<?
$emailerobject = new Emailer("hasin@pageflakes.com");
$emailerobject->addRecipients("hasin@somewherein.net");
```

```
$emailerobject->setSubject("To tylko test");
$emailerobject->setBody("Cześć Hasin,Jak się miewasz?");
$emailerobject->sendEmail();
?>
```

Na powyższym fragmencie kodu pierwszym krokiem jest utworzenie egzemplarza klasy `Emailer` i przypisania go zmiennej o nazwie `$emailerobject`. Warto w tym miejscu zapamiętać bardzo ważną regułę: podczas tworzenia nowego obiektu `Emailer` należy podać adres nadawcy. Cały wiersz jest więc następujący:

```
$emailerobject = new Emailer("hasin@pageflakes.com");
```

Wynika to z faktu, że metoda konstruktora jest zdefiniowana w postaci `__construct($sender)`. Jak wspomniano wcześniej, podczas tworzenia egzemplarza obiektu następuje automatyczne wywołanie konstruktora. Dlatego też w trakcie ustanawiania klasy `Emailer` trzeba podać prawidłowe argumenty, zgodnie z definicją zawartą w metodzie konstruktora. Przykładowo, wykonanie poniższego kodu spowoduje wygenerowanie komunikatu ostrzeżenia:

```
<?
$emailer = new emailer();
?>
```

Po wykonaniu powyższego kodu na ekranie zostanie wyświetlony komunikat ostrzeżenia:

```
Warning: Missing argument 1 for emailer::__construct(),
called in C:\OOP_PHP5\Kody\rozdzial1\class.emailer.php on line 42
and defined in <b>C:\OOP_PHP5\Kody\rozdzial1\class.emailer.php</b>
on line <b>9</b><br />
```

Teraz różnica powinna być doskonale widoczna. Jeżeli klasa nie posiada metody konstruktora bądź konstruktor nie zawiera argumentów, wtedy egzemplarz obiektu można utworzyć za pomocą powyższego kodu.

Modyfikatory dostępu

W omówionej wcześniej klasie zastosowano kilka słów kluczowych, między innymi `private` i `public`. Powstaje więc pytanie, co oznaczają te słowa kluczowe i dlaczego ich zastosowanie w klasie jest konieczne? Ogólnie rzecz biorąc, wymienione słowa kluczowe są nazywane modyfikatorami dostępu i zostały wprowadzone w PHP 5. Modyfikatory dostępu *nie występowały* w PHP 4. Te słowa kluczowe pomagają programiście w definiowaniu ograniczeń w dostępności do zmiennych i właściwości dla użytkowników danej klasy. Przekonajmy się, w jaki sposób można wykorzystać dostępne modyfikatory dostępu.

Private. Właściwości lub metody zadeklarowane z użyciem słowa kluczowego `private` (prywatne) nie mogą być wywołane z zewnątrz klasy. Jednocześnie dowolne metody wewnątrz tej

samej klasy mogą bez problemu uzyskać dostęp do elementów prywatnych. W omawianej klasie `Emailer` wszystkie właściwości zostały zdefiniowane jako prywatne, dlatego też wykonanie poniższego kodu spowoduje wygenerowanie komunikatu błędu:

```
<?
include_once("class.emailer.php");
$emobject = new Emailer("hasin@somewherein.net");
$emobject->subject = "Witaj świecie";
?>
```

Po wykonaniu powyższego kodu zostanie wygenerowany błąd krytyczny:

```
<b>Fatal error</b>: Cannot access private property emailer::$subject
 in <b>C:\OOP_PHP5\Kody\rozdzial1\class.emailer.php</b> on line
<b>43</><br />
```

Oznacza to, że z zewnątrz klasy nie można uzyskać dostępu do jakiejkolwiek prywatnej właściwości bądź metody.

Public. Każda właściwość lub metoda, która nie została wyraźnie zdefiniowana z użyciem słów kluczowych `private` (prywatna) bądź `protected` (chroniona), jest metodą publiczną (`public`). Dostęp do metod publicznych jest możliwy również z zewnątrz klasy.

Protected. To jest kolejny modyfikator dostępu, który ma znaczenie specjalne w programowaniu zorientowanym obiektowo. Jeżeli jakakolwiek właściwość lub metoda zostanie zdefiniowana z użyciem słowa kluczowego `protected`, to dostęp do niej można uzyskać tylko z poziomu podklasy. Więcej informacji dotyczących podklas zostanie przedstawionych w dalszej części rozdziału. Aby zademonstrować, w jaki sposób działa chroniona metoda lub właściwość, posłużymy się kolejnym przykładem.

Rozpoczynamy od otwarcia pliku *class.emailer.php* (czyli klasy `Emailer`) i zmieniamy deklarację zmiennej `$sender`. Po zmianie definicja zmiennej powinna być następująca:

```
protected $sender
```

Następnie tworzymy kolejny plik o nazwie *class.extendedemailer.php*, w którym powinien znajdować się poniższy fragment kodu:

```
<?
class ExtendedEmailer extends emailer
{
function  __construct(){}
   public function setSender($sender)
   {
      $this->sender = $sender;
   }
}
?>
```

Kolejny krok to użycie w następujący sposób nowo utworzonego obiektu:

```
<?
include_once("class.emailer.php");
include_once("class.extendedemailer.php");
$xemailer = new ExtendedEmailer();
$xemailer->setSender("hasin@pageflakes.com");
$xemailer->addRecipients("hasin@somewherein.net");
$xemailer->setSubject("To tylko test ");
$xemailer->setBody("Cześć Hasin,Jak się miewasz?");
$xemailer->sendEmail();
?>
```

Po dokładnym przyjrzeniu się kodowi klasy `ExtendedEmailer` czytelnik zauważy, że następuje próba uzyskania dostępu do właściwości `$sender` jej klasy nadrzędnej (którą w rzeczywistości jest klasa `Emailer`). Dostęp do wymienionej właściwości jest możliwy, ponieważ została zadeklarowana jako chroniona. Dodatkową zaletą jest fakt, że właściwość `$sender` nadal pozostaje bezpośrednio niedostępna poza zasięgiem obu wymienionych klas. Oznacza to, że próba wykonania poniższego fragmentu kodu spowoduje wygenerowanie błędu krytycznego:

```
<?
include_once("class.emailer.php");
include_once("class.extendedemailer.php");
$xemailer = new ExtendedEmailer();
$xemailer->sender = "hasin@pageflakes.com";
?>
```

Po wykonaniu powyższego kodu zostanie wygenerowany błąd krytyczny:

```
<b>Fatal error</b>:  Cannot access protected property
extendedEmailer::$sender in <b>C:\OOP_PHP5\Kody\rozdzial1\test.php
</b> on line <b>5</b><br />
```

Konstruktory i destruktory

We wcześniejszej części rozdziału wspomniano o metodzie konstruktora. Wymieniony konstruktor to metoda specjalna, która jest automatycznie wykonywana podczas tworzenia egzemplarza klasy. W języku PHP 5 istnieją dwa sposoby napisana metody konstruktora wewnątrz klasy. Pierwszy z nich to po prostu zdefiniowanie w klasie metody o nazwie `__construct()`. Natomiast drugim sposobem jest utworzenie metody o nazwie identycznej jak nazwa klasy. Przykładowo, jeśli klasa nosi nazwę `Emailer`, to nazwą metody konstruktora będzie `Emailer()`. Przyjrzyjmy się poniższej klasie, której zadaniem jest obliczanie silni dowolnej liczby:

```
<?
// class.factorial.php
class factorial
```

```
{
   private $result = 1;// Inicjalizację można przeprowadzić bezpośrednio z zewnątrz.
   private $number;
   function __construct($number)
   {
      $this->number = $number;
      for($i=2; $i<=$number; $i++)
      {
         $this->result *= $i;
      }
   }
   public function showResult()
   {
      echo "Silnia liczby {$this->number} wynosi {$this->result}. ";
   }
}
?>
```

Na powyższym fragmencie kodu do zdefiniowania konstruktora wykorzystano metodę `__construct()`. Działanie kodu pozostanie bez zmian, jeśli nazwa metody `__construct()` zostanie zmieniona na `factorial()`.

W tym miejscu może zrodzić się pytanie, czy w klasie dopuszczalne jest użycie konstruktorów zdefiniowanych za pomocą obu omówionych stylów? Oznacza to istnienie w klasie funkcji o nazwie `__construct()` oraz funkcji o nazwie identycznej z nazwą klasy. Który z konstruktorów zostanie użyty w takim przypadku? A może zostaną wykonane obie te funkcje? To są bardzo trafne i ciekawe pytania. Warto zapamiętać, że w rzeczywistości jednak nie ma możliwości wykonania obu funkcji. Jeżeli w klasie będą zdefiniowane dwie metody konstruktora, to PHP 5 daje pierwszeństwo funkcji `__construct()`, natomiast druga metoda konstruktora będzie zignorowana. Spójrzmy na poniższy fragment kodu:

```
<?
// class.factorial.php
class Factorial
{
   private $result = 1;
   private $number;
   function __construct($number)
   {
      $this->number = $number;
      for($i=2; $i<=$number; $i++)
      {
         $this->result*=$i;
      }
      echo "Wykonano metodę __construct(). ";
   }
   function factorial($number)
   {
      $this->number = $number;
```

```
        for($i=2; $i<=$number; $i++)
        {
           $this->result*=$i;
        }
        echo " Wykonano metodę factorial(). ";
     }
     public function showResult()
     {
        echo " Silnia liczby {$this->number} wynosi {$this->result}. ";
     }
}
?>
```

Jeżeli powyższa klasa zostanie użyta w następujący sposób:

```
<?
include_once("class.factorial.php");
$fact = new Factorial(5);
$fact->showResult();
?>
```

to na ekranie zostanie wyświetlony poniższy komunikat:

```
Wykonano metodę __construct().Silnia liczby 5 wynosi 120.
```

Podobnie do metody konstruktora w klasie występuje również metoda destruktora, która jest wykonywana w trakcie niszczenia obiektu. Programista może wyraźnie utworzyć destruktora poprzez zdefiniowanie metody o nazwie `__destruct()`. Wymieniona metoda zostanie automatycznie wywołana przez PHP na samym końcu wykonywania danego skryptu. Aby sprawdzić, w jaki sposób działa destruktor, można w omówionej powyżej klasie dodać metodę destruktora:

```
function __destruct()
{
   echo "Obiekt został zniszczony.";
}
```

Następnie, po ponownym wykonaniu skryptu obliczającego silnię, na ekranie zostanie wyświetlony następujący komunikat:

```
Wykonano metodę __construct(). Silnia liczby 5 wynosi 120. Obiekt został
zniszczony.
```

Stałe klasy

Czytelnik prawdopodobnie wie, że w skryptach PHP definiowanie stałej odbywa się za pomocą słowa kluczowego `define` (definiowanie nazwy stałej oraz jej wartości). Jednak w celu zdefiniowania stałej w klasie używa się słowa kluczowego `const`. W rzeczywistości te stałe

funkcjonują na zasadzie zmiennych statycznych, a jedyna różnica między nimi polega na tym, że są tylko do odczytu. Przykład tworzenia i używania stałych w klasie został przedstawiony na poniższym fragmencie kodu:

```
<?
class WordCounter
{
   const ASC=1;  // Przed stałą nie trzeba stosować znaku dolara ($).
   const DESC=2;
   private $words;
   function __construct($filename)
   {
      $file_content = file_get_contents($filename);
      $this->words =
         (array_count_values(str_word_count(strtolower
            ($file_content),1)));
   }
   public function count($order)
   {
      if ($order==self::ASC)
      asort($this->words);
      else if($order==self::DESC)
      arsort($this->words);
      foreach ($this->words as $key=>$val)
      echo $key ." = ". $val."<br/>";
   }
}
?>
```

Powyższa klasa `WordCounter` powoduje zliczanie częstotliwości występowania słów w podanym pliku. W kodzie zdefiniowano dwie stałe `ASC` i `DESC` o wartościach odpowiednio `1` i `2`. Aby wewnątrz klasy uzyskać dostęp do stałej, należy odnieść się do niej za pomocą słowa kluczowego `self`. Warto zwrócić uwagę, że dostęp do stałej następuje za pomocą operatora `::`, a nie operatora `->`. Wynika to z faktu, że stałe działają na zasadzie podobnej do elementów statycznych.

W celu użycia powyższej klasy trzeba wykorzystać przedstawiony poniżej fragment kodu. Zaprezentowano w nim również sposób dostępu do stałej:

```
<?
include_once("class.wordcounter.php");
$wc = new WordCounter("words.txt");
$wc->count(WordCounter::DESC);
?>
```

Warto zwrócić uwagę, że dostęp do stałej klasy następuje z zewnątrz klasy za pomocą operatora `::` umieszczonego tuż za nazwą klasy, a nie za nazwą egzemplarza klasy. Kolejnym krokiem, który trzeba wykonać w celu przetestowania omówionego skryptu, jest utworzenie pliku tekstowego *words.txt*. Wymieniony plik musi znajdować się w tym samym katalogu, w którym umieszczono skrypt:

Plik: words.txt

```
Wordpress jest silnikiem bloga dostepnym na licencji open source. Czytelnikom
nieznajacym blogów wyjasniamy, ze blog pozwala uzytkownikowi na prowadzenie
dziennika w Internecie. Wordpress jest zupelnie bezplatny i zostal wydany
na licencji GPL.
```

Po wykonaniu skryptu z podanym powyżej plikiem zostaną wyświetlone następujące dane wyjściowe:

```
na = 3
licencji = 2
wordpress = 2
blog = 2
w = 2
jest = 2
internecie = 1
dziennika = 1
prowadzenie = 1
zupelnie = 1
bezplatny = 1
gpl = 1
wydany = 1
zostal = 1
i = 1
uzytkownikowi = 1
pozwala = 1
source = 1
open = 1
bloga = 1
czytelnikom = 1
nieznajacym = 1
ze = 1
wyjasniamy = 1
silnikiem = 1
dostepnym = 1
```

Użyteczny skrypt, nieprawdaż?

Rozszerzanie klasy (dziedziczenie)

Jedną z najistotniejszych funkcji programowania zorientowanego obiektowo jest możliwość rozszerzenia klasy oraz utworzenie zupełnie nowego obiektu. Ten nowo utworzony obiekt będzie posiadał wszystkie funkcje obiektu nadrzędnego, które będą mogły zostać rozbudowane bądź nadpisane. Nowy obiekt może także zawierać zupełnie nowe funkcje. Na poniższym fragmencie kodu rozszerzono przedstawioną wcześniej klasę `Emailer` oraz nadpisano funkcję `sendEmail`, która obecnie ma możliwość wysyłania wiadomości e-mail w formacie HTML.

```
<?
class HtmlEmailer extends emailer
{
   public function sendHTMLEmail()
   {
         foreach ($this->recipients as $recipient)
         {
            $headers  = 'MIME-Version: 1.0' . "\r\n";
            $headers .= 'Content-type: text/html; charset=iso-8859-2' .
               "\r\n";
            $headers .= 'From: {$this->sender}' . "\r\n";
            $result = mail($recipient, $this->subject, $this->body,
               $headers);
            if ($result) echo "Wiadomość w formacie HTML została wysłana do
               {$recipient}<br/>";
      }
   }
}
?>
```

Ponieważ nowa klasa rozszerza klasę `Emailer` oraz wprowadza nową funkcję o nazwie `sendHTMLEmail()`, to programista nadal posiada dostęp do wszystkich metod obecnych w klasie nadrzędnej. Oznacza to, że przedstawiony poniżej fragment kodu jest jak najbardziej prawidłowy:

```
<?
include_once("class.htmlemailer.php");
$hm = new HtmlEmailer();
// ...miejsce na inne zadania...
$hm->sendEmail();
$hm->sendHTMLEmail();
?>
```

Jeżeli zachodzi potrzeba uzyskania dostępu do dowolnej metody klasy nadrzędnej (inaczej nazywanej superklasą), to można użyć słowa kluczowego `parent`. Przykładowo, jeżeli programista chce uzyskać dostęp do metody o nazwie `sayHello`, to należy wydać polecenie `parent::sayHello();`.

Warto zwrócić uwagę, że w klasie `HtmlEmailer` nie została zdefiniowana funkcja o nazwie `sendEmail()`. Natomiast wymieniona metoda działa z klasy nadrzędnej, czyli `Emailer`.

> W omówionym powyżej przykładzie klasa `HtmlEmailer` jest podklasą klasy `Emailer`, natomiast klasa `Emailer` to superklasa dla klasy `HtmlEmailer`. Trzeba zapamiętać, że jeśli podklasa nie posiada konstruktora, to zostanie użyta metoda konstruktora klasy nadrzędnej. W trakcie pisania niniejszej książki wielokrotne dziedziczenie na poziomie klasy nie było obsługiwane. Oznacza to, że nie można jednocześnie dziedziczyć z więcej niż tylko jednej klasy. Jednak wielokrotne dziedziczenie jest obsługiwane w interfejsach. Dlatego też interfejs może rozszerzać dowolną liczbę interfejsów.

Nadpisywanie metod

W rozszerzanym obiekcie można nadpisać dowolną metodę (zdefiniowaną jako chronioną lub publiczną) i dowolnie zmieniać sposób jej działania. W jaki więc sposób można nadpisać dowolną metodę? Wystarczy po prostu utworzyć funkcję o takiej samej nazwie jak ta, która ma zostać nadpisana. Przykładowo, po utworzeniu w klasie `HtmlEmailer` funkcji o nazwie `sendEmail` spowoduje ona nadpisanie metody `sendEmail()` zdefiniowanej w klasie nadrzędnej `Emailer`. Jeżeli w podklasie zostanie zdefiniowana zmienna, która istnieje także w superklasie, to podczas dostępu do zmiennej zostanie użyta ta zdefiniowana w podklasie.

Uniemożliwianie nadpisywania

Jeżeli metoda zostanie zdefiniowana z użyciem słowa kluczowego `final`, to nie będzie mogła zostać nadpisana w żadnej podklasie. Dlatego też, jeżeli programista nie chce, aby dana metoda była nadpisywana, to wystarczy zdefiniować ją jako `final`. Poniżej pokazano definicję metody z użyciem słowa kluczowego `final`:

```
<?
class SuperClass
{
   public final function someMethod()
   {
      // ...miejsce na dowolny kod...
   }
}
class SubClass extends SuperClass
{
   public function someMethod()
   {
      // ...miejsce na dowolny kod, ale i tak nie zostanie on wykonany...
   }
}
?>
```

Jeżeli powyższy kod zostanie wykonany, to spowoduje wygenerowanie błędu krytycznego, ponieważ klasa `SubClass` próbuje nadpisać metodę z klasy nadrzędnej `SuperClass`, która została zdefiniowana z użyciem słowa kluczowego `final`.

Uniemożliwianie rozszerzania

Podobnie jak w przypadku metody zdefiniowanej jako `final`, także klasę można zdefiniować z użyciem słowa kluczowego `final`, które uniemożliwi jej rozszerzanie. Dlatego też po zdefiniowaniu klasy w sposób przedstawiony na poniższym listingu nie będzie można jej dalej rozszerzać:

```
<?
final class aclass
{
}
class bclass extends aclass
{
}
?>
```

Po wykonaniu powyższego kodu zostanie wygenerowany następujący błąd krytyczny:

```
<b>Fatal error</b>:  Class bclass may not inherit from final class
(aclass) in <b>C:\OOP_PHP5\Kody\rozdzial1\class.aclass.php</b> on
line <b>8</b><br />
```

Polimorfizm

Jak już wspomniano we wcześniejszej części książki, polimorfizm jest procesem tworzenia kilku obiektów z określonych klas bazowych. Przykładowo, warto spojrzeć na poniższy przykład, w którym wykorzystano wszystkie trzy klasy omówione dotychczas w rozdziale `Emailer`, `ExtendedEmailer` oraz `HtmlEmailer`:

```
<?
include("class.emailer.php");
include("class.extendedemailer.php");
include("class.htmlemailer.php");
$emailer = new Emailer("hasin@somewherein.net");
$extendedemailer = new ExtendedEmailer();
$htmlemailer = new HtmlEmailer("hasin@somewherein.net");
if ($extendedemailer instanceof emailer)
echo "Klasa Extended Emailer wywodzi się z klasy Emailer.<br/>";
if ($htmlemailer instanceof emailer)
echo "Klasa HTML Emailer również wywodzi się z klasy Emailer.<br/>";
if ($emailer instanceof htmlEmailer)
echo "Klasa Emailer wywodzi się z klasy HTMLEmailer.<br/>";
if ($htmlemailer instanceof extendedEmailer)
echo "Klasa HTML Emailer wywodzi się z klasy Emailer.<br/>";
?>
```

Po wykonaniu powyższego fragmentu kodu zostaną wyświetlone następujące dane wyjściowe:

```
Klasa Extended Emailer wywodzi się z klasy Emailer.
Klasa HTML Emailer również wywodzi się z klasy Emailer.
```

To jest przykład polimorfizmu.

Dzięki zastosowaniu operatora `instanceof` zawsze istnieje możliwość sprawdzenia, czy klasa wywodzi się z innej klasy.

Interfejs

Interfejs jest pustą klasą, która zawiera jedynie deklaracje metod. Dlatego też każda klasa implementująca dany interfejs musi zawierać deklaracje zawartych w nim funkcji. Interfejs jest więc jedynie zbiorem ściśle określonych reguł, które pomagają w rozszerzaniu dowolnej klasy oraz ścisłej implementacji wszystkich metod zadeklarowanych w interfejsie. Klasa może stosować dowolny interfejs, używając słowa kluczowego `implements`. Warto zwrócić uwagę, że w interfejsie można jedynie zadeklarować metody, ale nie można umieścić w nim definicji tychże metod. Oznacza to, że w interfejsie części główne wszystkich metod pozostają puste.

Powstaje więc pytanie, do czego może służyć interfejs? Jednym z powodów jego stosowania jest możliwość implementacji ściśle określonych reguł podczas definicji klasy. Przykładowo, programista wie, że musi utworzyć klasy pewnego sterownika dla programu, które będą zawierały operacje związane z bazą danych. Dla bazy danych MySQL będzie to jedna klasa, dla PostgreSQL będzie to kolejna klasa, dla SQLite kolejna, itd. W takim przypadku zespół programistów może liczyć trzy osoby, z których każda będzie oddzielnie tworzyła wskazaną klasę.

Można teraz zadać sobie pytanie, jaki byłby wynik pracy tych programistów, gdyby każdy z nich implementował w klasie własny styl? Inni programiści, którzy chcieliby wykorzystać te klasy sterowników, musieliby poznać definicje użytych metod, a następnie stosować taki sam styl, aby móc je wykorzystać we własnym kodzie. Takie rozwiązanie staje się wyjątkowo trudne w obsłudze. Dlatego też można po prostu ustalić, że każda klasa sterownika musi posiadać dwie metody o nazwach `connect()` i `execute()`. W takim przypadku programiści nie muszą przejmować się wewnętrzną strukturą sterownika, ponieważ doskonale wiedzą, że wszystkie klasy posiadają takie same definicje metod. Interfejs stanowi więc duże ułatwienie podczas pracy nad tego rodzaju projektem. Poniżej przedstawiono kod przykładowego interfejsu:

```
<?
// interface.dbdriver.php
interface DBDriver
{
   public function connect();
   public function execute($sql);
}
?>
```

Czy czytelnik zwrócił uwagę na fakt, że w interfejsie definicje funkcji są puste? Kolejny krok to utworzenie klasy `MySQLDriver`, która będzie implementowała przedstawiony powyżej interfejs:

```
<?
// class.mysqldriver.php
include("interface.dbdriver.php");
class MySQLDriver implements DBDriver
{
}
?>
```

Jeżeli powyższy kod zostanie uruchomiony, to spowoduje wygenerowanie poniższego komunikatu błędu. Wynika to z faktu, że w klasie `MySQLDriver` nie zostały zdefiniowane funkcje `connect()` i `execute()`, które są zadeklarowane w interfejsie. Warto więc uruchomić kod i odczytać komunikat błędu:

```
<b>Fatal error</b>: Class MySQLDriver contains 2 abstract methods
and must therefore be declared abstract or implement the remaining
methods (DBDriver::connect, DBDriver::execute) in
<b>C:\OOP_PHP5\Kody\rozdzial1\class.mysqldriver.php</b> on line <b>5</b><br />
```

Kolejny krok to dodanie do klasy `MySQLDriver` dwóch metod. Po wprowadzeniu zmian kod przedstawia się następująco:

```
<?
include("interface.dbdriver.php");
class MySQLDriver implements DBDriver
{
   public function connect()
   {
      // Nawiązanie połączenia z bazą danych.
   }
   public function execute()
   {
      // Wykonanie zapytania i wyświetlenie jego wyników.
   }
}
?>
```

Po uruchomieniu powyższego kodu na ekranie ponownie zostanie wyświetlony komunikat błędu:

```
<b>Fatal error</b>:  Declaration of MySQLDriver::execute() must be
compatible with that of DBDriver::execute() in
<b>C:\OOP_PHP5\Kody\rozdzial1\class.mysqldriver.php</b> on line <b>3</b><br />
```

Ten komunikat informuje użytkownika, że metoda `execute()` nie jest zgodna ze strukturą metody `execute()`, która została zadeklarowana w interfejsie. Po dokładnym przyjrzeniu się interfejsowi czytelnik zauważy, że metoda `execute()` powinna posiadać jeden argument. Oznacza to, że w trakcie implementacji interfejsu w tworzonych klasach każda struktura metody musi być dokładnie taka sama jak zadeklarowana w interfejsie. Po przepisaniu klasy `MySQLDriver` jej kod przedstawia się następująco:

```
<?
include("interface.dbdriver.php");
class MySQLDriver implements DBDriver
{
   public function connect()
   {
      // Nawiązanie połączenia z bazą danych.
   }
   public function execute($query)
   {
      // Wykonanie zapytania i wyświetlenie jego wyników.
   }
}
?>
```

Klasa abstrakcyjna

Klasa abstrakcyjna jest niemal taką samą konstrukcją jak interfejs, za wyjątkiem faktu, że deklarowane w niej metody mogą posiadać definicje. Ponadto klasa abstrakcyjna musi być „rozszerzana", a nie „implementowana". Dlatego też, jeżeli rozszerzane klasy posiadają metody o takich samych funkcjach, to te funkcje można zdefiniować w klasie abstrakcyjnej. Poniżej przedstawiono przykład klasy abstrakcyjnej:

```
<?
// abstract.reportgenerator.php
abstract class ReportGenerator
{
   public function generateReport($resultArray)
   {
      // Miejsce na kod przetwarzający wielowymiarową tablicę wynikową oraz
      // generujący raport w postaci kodu HTML.
   }
}
?>
```

W powyższej klasie abstrakcyjnej znajduje się metoda o nazwie `generateRaport`, która jako argument pobiera wielowymiarową tablicę, a następnie na jej podstawie generuje raport w postaci kodu HTML. Powstaje zatem pytanie, dlaczego ta metoda została umieszczona w klasie abstrakcyjnej? Odpowiedź jest prosta — ponieważ generowanie raportu będzie wspólną funkcją wszystkich sterowników baz danych. Sama funkcja nie wpływa również na kod sterownika, ponieważ jako argument pobiera tablicę i nie ma nic wspólnego z bazą danych. Dlatego też w przedstawionym poniżej kodzie klasy `MySQLDriver` zastosowano klasę abstrakcyjną. Warto zwrócić uwagę, że cały kod odpowiedzialny za generowanie raportu został już wcześniej napisany. Nie trzeba więc umieszczać go ponownie w klasie sterownika, jak miałoby to miejsce w przypadku interfejsu.

```
<?
include("interface.dbdriver.php");
include("abstract.reportgenerator.php");
class MySQLDriver extends ReportGenerator implements DBDriver
{
   public function connect()
   {
      // Nawiązanie połączenia z bazą danych.
   }
   public function execute($query)
   {
      // Wykonanie zapytania i wyświetlenie jego wyników.
   }
   // Nie trzeba w tym miejscu ponownie definiować lub umieszczać metody generateReport
   // ponieważ ta klasa bezpośrednio rozszerza klasę abstrakcyjną.
}
?>
```

Warto zwrócić uwagę, że jednocześnie można zarówno używać klasy abstrakcyjnej, jak i implementować interfejs. Zostało to przedstawione na powyższym fragmencie kodu.

> Klasę abstrakcyjną (`abstract`) nie można zdefiniować za pomocą słowa kluczowego `final`, ponieważ klasa abstrakcyjna musi być rozszerzana. Natomiast słowo kluczowe `final` uniemożliwia rozszerzenie tak zdefiniowanej klasy. Dlatego też jednoczesne użycie tych dwóch wymienionych słów kluczowych jest bezsensowne i język PHP na to nie pozwala.

Oprócz zdefiniowania klasy jako abstrakcyjnej także i metodę można zdefiniować z użyciem słowa kluczowego `abstract`. Zdefiniowanie metody abstrakcyjnej oznacza, że podklasy muszą nadpisywać tę metodę. W deklaracji metody abstrakcyjnej nie powinna znajdować się jej definicja. Przykład deklaracji metody abstrakcyjnej został przedstawiony poniżej:

```
abstract public function connectDB();
```

Metody i właściwości statyczne

Słowo kluczowe `static` jest istotne w programowaniu zorientowanym obiektowo. Metody i właściwości statyczne pełnią bardzo ważną rolę zarówno w projekcie programu, jak i wzorcach projektowych. Czym więc są metody i właściwości statyczne?

Jak wcześniej przedstawiono, w celu uzyskania dostępu do dowolnej metody bądź atrybutu klasy wcześniej trzeba utworzyć jej egzemplarz (na przykład za pomocą słowa kluczowego `new`, czyli `$object = new emailer()`). W przeciwnym razie nie będzie można uzyskać dostępu do metod i właściwości danej klasy. Istnieje jednak odstępstwo od tej reguły i dotyczy metod i właściwości statycznych. Do metody lub właściwości statycznej programista może uzyskać

dostęp bezpośrednio bez potrzeby tworzenia egzemplarza danej klasy. Element statyczny jest więc podobny do elementu globalnego danej klasy i wszystkich jej egzemplarzy. Ponadto właściwości statyczne zachowują stan z ostatniego przypisania, co w niektórych sytuacjach jest bardzo użyteczne.

Czytelnik może zadać pytanie, dlaczego ktokolwiek chciałby używać metod statycznych? Cóż, są one bardzo podobne do metod pomocniczych. Wykonują więc ściśle określone zadanie lub zwracają ściśle określony obiekt. (Właściwości i metody statyczne są intensywnie używane we wzorcach projektowych, co zostanie przedstawione w dalszej części rozdziału). Z tego powodu deklarowanie nowego obiektu za każdym razem do wykonania takiego zadania może być uznane za niepotrzebne zużywanie zasobów. Spójrzmy więc na przykład użycia metod statycznych.

Wróćmy do omawianego wcześniej programu, który zajmuje się obsługą trzech baz danych — MySQL, PostgreSQL i SQLite. Zakładamy, że w danej chwili zachodzi potrzeba używania tylko jednego sterownika. W tym celu tworzymy klasę `DBManager`, której zadaniem jest utworzenie egzemplarza dowolnego sterownika oraz jego zwrócenie programiście.

```
<?
// class.dbmanager.php
class DBManager
{
   public static function getMySQLDriver()
   {
      // Utworzenie nowego egzemplarza obiektu sterownika bazy danych MySQL i jego
      // zwrócenie.
   }
   public static function getPostgreSQLDriver()
   {
      // Utworzenie nowego egzemplarza obiektu sterownika bazy danych PostgreSQL i jego
      // zwrócenie.
   }
   public static function getSQLiteDriver()
   {
      // Utworzenie nowego egzemplarza obiektu sterownika bazy danych SQLite i jego
      // zwrócenie.
   }
}
?>
```

W jaki sposób można użyć powyższą klasę? Dostęp do dowolnej właściwości statycznej odbywa się poprzez operator `::` zamiast operatora `->`. Przykład użycia klasy `DBManager` został przedstawiony poniżej:

```
<?
// test.dbmanager.php
include_once("class.dbmanager.php");
$dbdriver = DBManager::getMySQLDriver();
// Miejsce na kod przetwarzający operacje bazy danych za pomocą obiektu $dbdriver.
?>
```

Warto zwrócić uwagę, że w kodzie nie następuje tworzenie nowego egzemplarza obiektu `DBManager`, na przykład za pomocą polecenia `$dbmanager = new DBManager()`. Zamiast tego, używając operatora `::` programista uzyskuje bezpośredni dostęp do jednej z metod wymienionego obiektu.

Co zyskuje programista, stosując tego typu rozwiązanie? Ogólnie rzecz biorąc, skoro po prostu potrzebny jest obiekt sterownika, to nie ma potrzeby tworzenia nowego obiektu `DBManager` i zużywania przez niego pamięci aż do chwili zakończenia działania skryptu. Metoda statyczna zwykle wykonuje swoje zadanie, a następnie kończy działanie.

Trzeba zapamiętać jedną bardzo ważną kwestię. Wewnątrz metody statycznej nie można używać pseudoobiektu `$this`. Ponieważ nie jest tworzony egzemplarz klasy, to słowo kluczowe `$this` nie istnieje wewnątrz metody statycznej. Zamiast niego należy stosować słowo kluczowe `self`.

Spójrzmy na poniższy fragment kodu, w którym zademonstrowano rzeczywisty sposób działania właściwości statycznej:

```
<?
// class.statictester.php
class StaticTester
{
   private static $id=0;
   function __construct()
   {
      self::$id +=1;
   }
   public static function checkIdFromStaticMehod()
   {
      echo "Bieżące Id z metody statycznej wynosi ".self::$id."\n";
   }
   public function checkIdFromNonStaticMethod()
   {
      echo " Bieżące Id z metody niestatycznej wynosi ".self::$id."\n";
   }
}
$st1 = new StaticTester();
StaticTester::checkIdFromStaticMehod();
$st2 = new StaticTester();
$st1->checkIdFromNonStaticMethod(); // Zwrot wartości $id jako 2.
$st1->checkIdFromStaticMehod();
$st2->checkIdFromNonStaticMethod();
$st3 = new StaticTester();
StaticTester::checkIdFromStaticMehod();
?>
```

Po uruchomieniu powyższego kodu zostaną wyświetlone następujące dane wyjściowe:

```
Bieżące Id z metody statycznej wynosi 1
Bieżące Id z metody niestatycznej wynosi 2
Bieżące Id z metody statycznej wynosi 2
Bieżące Id z metody niestatycznej wynosi 2
Bieżące Id z metody statycznej wynosi 3
```

Kiedy tylko zostanie utworzony nowy egzemplarz obiektu, będzie on wpływał na pozostałe egzemplarze, ponieważ zmienna została zdefiniowana jako statyczna. Używanie tej możliwości, czyli specjalnego wzorca projektowego o nazwie „Singleton", doskonale sprawdza się w PHP.

Ostrzeżenie dotyczące używania elementów statycznych

Elementy statyczne powodują, że obiekty zachowują się w sposób podobny do proceduralnego stylu działania. Bez tworzenia egzemplarzy programista może bezpośrednio wywołać dowolną funkcję, podobnie jak w programowaniu proceduralnym. Z tego powodu metody statyczne powinny być używane z zachowaniem ostrożności. Nadmierne korzystanie z metod statycznych jest nieużyteczne. O ile nie zachodzi taka konieczność, to należy unikać używania elementów statycznych.

Metody akcesorów

Metody akcesorów to po prostu metody, których zadaniem jest pobieranie i ustalanie wartości dowolnej właściwości klasy. Dobrym nawykiem jest uzyskiwanie dostępu do właściwości klasy za pomocą metod akcesorów zamiast bezpośredniego ustalania lub pobierania ich wartości. Chociaż metody akcesorów są takie same jak inne metody, to jednak istnieją pewne konwencje ich tworzenia.

Dostępne są dwa rodzaje metod akcesorów. Pierwszy z nich nosi nazwę `getter`, a celem tej metody jest pobranie wartości dowolnej właściwości klasy. Drugi rodzaj metody nosi nazwę `setter` i służy do ustalania wartości dowolnej właściwości klasy. Poniżej zaprezentowano przykładowe metody `getter` i `setter` używane do operacji na właściwościach klasy:

```
<?
class Student
{
   private $name;
   private $roll;
   public function setName($name)
   {
      $this->name= $name;
   }
   public function setRoll($roll)
   {
      $this->roll =$roll;
   }
```

```
    public function getName()
    {
        return $this->name;
    }
    public function getRoll()
    {
        return $this->roll;
    }
}
?>
```

W powyższym fragmencie kodu zastosowano po dwie metody typu `getter` oraz `setter`. To jest konwencja pisania metod akcesorów. Metoda typu `setter` powinna rozpoczynać się słowem kluczowym `set`, a następnie zawierać nazwę właściwości, której pierwsza litera jest duża. Podobnie, metoda typu `getter` powinna rozpoczynać się słowem kluczowym `get`, a następnie zawierać nazwę zmiennej, w której pierwsza litera jest duża. Oznacza to, że jeśli nazwą właściwości jest `email`, to metoda typu `getter` powinna być nazwana `getEmail`, natomiast metoda typu `setter` powinna mieć nazwę `setEmail`. I to tyle!

Czytelnik może w tym miejscu zapytać, dlaczego ktokolwiek mógłby chcieć wykonywać dodatkową pracę, definiując te metody, skoro zmienne można zdefiniować jako publiczne, a resztę pozostawić bez zmian? Czy to nie będzie miało takiego samego efektu? Ogólnie rzecz ujmując, nie. Używając metod akcesorów, programista otrzymuje dodatkowe korzyści. Przede wszystkim zachowuje pełną kontrolę nad ustalaniem i pobieraniem wartości dowolnej właściwości. „I co z tego?" — mógłby zapytać czytelnik. Załóżmy, że zachodzi potrzeba zastosowania filtrów danych wejściowych użytkownika przed ustawieniem wartości właściwości. W takim przypadku metoda typu `setter` pozwala na filtrowanie danych wejściowych przed ich ustawieniem i użyciem w programie.

Czy jeśli w klasie znajduje się 100 właściwości, to programista musi napisać po sto metod typu `getter` i `setter`? To bardzo dobre pytanie. Język PHP jest na tyle elegancki, że wyręcza programistę z takiego żmudnego zadania. W jaki sposób? Odpowiedź na to pytanie znajduje się w kolejnym podrozdziale, w którym zostaną omówione metody magiczne służące do dynamicznego pobierania i ustalania wartości właściwości. Używanie tego rodzaju metod powoduje redukcję o około 90% pracy związanej z koniecznością żmudnego pisania kodu metod akcesorów. Aż trudno w to uwierzyć, nieprawdaż? Jeśli tak, to warto się o tym przekonać samodzielnie.

Używanie metod magicznych do pobierania i ustalania wartości właściwości klasy

Jak wspomniano w poprzednim podrozdziale, pisanie dużej liczby metod akcesorów dla właściwości klasy może być prawdziwym koszmarem. Aby uniknąć tego nudnego zadania, można wykorzystać metody magiczne. Taki proces nosi nazwę przeciążania metody.

W PHP 5 wprowadzono w klasach kilka metod magicznych, które znacznie ułatwiają pracę w niektórych zadaniach wykonywanych w OOP. Dwie z tych metod służą do dynamicznego pobierania i ustalania wartości w klasie. Wspomniane metody noszą nazwy `__get()` oraz `__set()`. Przykład ich użycia został przedstawiony w poniższym fragmencie kodu:

```
<?
// class.student.php
class Student
{
   private $properties = array();
   function __get($property)
   {
      return $this->properties[$property];
   }
   function __set($property, $value)
   {
      $this->properties[$property]="AutoSet {$property} jako: ".$value;
   }
}
?>
```

Kolejny krok to użycie tego kodu w programie. Powyższa klasa zostaje więc zastosowana w poniższym skrypcie:

```
<?
$st = new Student();
$st->name = "Afif";
$st->roll=16;
echo $st->name."\n";
echo $st->roll;
?>
```

Po wykonaniu kodu PHP natychmiast rozpozna, że w klasie nie istnieją właściwości `name` i `roll`. Ponieważ nazwy właściwości istnieją, to nastąpi wywołanie metody `__set()`, która następnie przypisze wartość nowo utworzonej właściwości klasy. Na ekranie zostaną więc wyświetlone następujące dane wyjściowe:

```
AutoSet name jako: Afif
AutoSet roll jako: 16
```

Wygląda to całkiem interesująco, nieprawdaż? Używając metod magicznych, programista wciąż zachowuje pełną kontrolę nad ustawianiem i pobieraniem wartości właściwości klasy. Stosowanie metod magicznych wiąże się jednak z jednym ograniczeniem. Podczas używania Reflection API nie ma możliwości badania właściwości klasy (wymienione Reflection API zostanie przedstawione w jednym z kolejnych rozdziałów). Ponadto, sama klasa traci nieco ze swojej „czytelności" oraz „łatwości obsługi". Dlaczego? Warto spojrzeć na poprzednią i nową klasę `Student`, aby samodzielnie odpowiedzieć sobie na to pytanie.

Metody magiczne służące do przeciążania metod klasy

Podobnie jak w przypadku przeciążania i używania metod akcesorów dostępne są również metody magiczne służące do przeciążania wywołania dowolnej metody klasy. Jeżeli czytelnik nadal nie rozumie pojęcia przeciążania metody, to warto przypomnieć, że jest to proces uzyskiwania dostępu do dowolnej metody, która nawet nie istnieje w klasie. Brzmi niewiarygodnie, nieprawdaż? Przyjrzyjmy się bliżej temu zagadnieniu.

Istnieje metoda magiczna, która pomaga w przeciążaniu dowolnego wywołania metody w kontekście klasy języka PHP 5. Nazwa tej metody magicznej to `__call()`. Pozwala ona na zdefiniowanie działań lub wartości zwrotnej w sytuacji, gdy w obiekcie następuje wywołanie niezdefiniowanej metody. Może to być używane do symulowania przeciążania metody lub nawet zapewnienia eleganckiej obsługi błędów, gdy niezdefiniowana metoda jest wywoływana w obiekcie. Metoda `__call()` pobiera dwa argumenty — nazwę metody oraz tablicę argumentów przekazywanych niezdefiniowanej metodzie.

Poniżej przedstawiono przykład użycia metody `__call()`:

```
<?
class Overloader
{
   function __call($method, $arguments)
   {
      echo "Wywołano metodę o nazwie {method} z następującymi
         argumentami <br/>";
      print_r($arguments);
      echo "<br/>";
   }
}
$ol = new Overloader();
$ol->access(2,3,4);
$ol->notAnyMethod("boo");
?>
```

Jak widać w powyższym kodzie, w klasie nie ma definicji metod `access` oraz `notAnyMethod`. Dlatego też próba ich wywołania powinna zakończyć się wygenerowaniem komunikatu błędu, nieprawdaż? Jednak technika przeciążania metody pomaga w sytuacji, gdy następuje wywołanie nieistniejącej metody. Po wykonaniu powyższego kodu czytelnik otrzyma następujące dane wyjściowe:

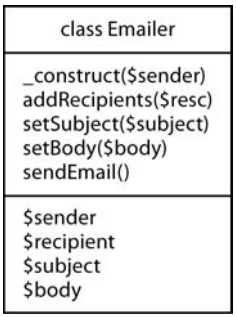
```
Wywołano metodę o nazwie access z następującymi argumentami
Array
(
    [0] => 2
```

```
    [1] => 3
    [2] => 4
)
Wywołano metodę o nazwie notAnyMethod z następującymi argumentami
Array
(
    [0] => boo
)
```

Oznacza to, że wszystkie argumenty zostały przekazane w postaci tablicy. Istnieje znacznie więcej metod magicznych, a niektóre z nich zostaną przedstawione w dalszej części książki.

Wizualne przedstawienie klasy

W programowaniu zorientowanym obiektowo czasami zachodzi potrzeba wizualnego przedstawienia klasy. Przekonajmy się więc, w jaki sposób można to zrobić. Na potrzeby tego zadania zostanie użyta klasa `Emailer` (zobacz rysunek 2.1):



Rysunek 2.1. Wizualne przedstawienie klasy

Na pokazanym rysunku możemy wyodrębnić trzy sekcje. Na samej górze znajduje się sekcja z nazwą klasy. W środku widzimy zapisane wszystkie metody zawierające bądź nie zawierające parametry. Najniższa sekcja pokazuje wszystkie właściwości klasy. I to tyle!

Podsumowanie

W rozdziale zostały omówione zagadnienia związane z tworzeniem obiektów oraz ich wzajemnym współdziałaniem. W porównaniu do PHP 4, język PHP w wersji 5 przynosi zadziwiające usprawnienia w zakresie modelu obiektowego. Silnik Zend Engine 2 stanowiący jądro PHP 5 jest również bardzo efektywny w obsłudze tych funkcji i pozwala na doskonałą optymalizację.

W następnym rozdziale zostaną szczegółowo przedstawione podstawowe funkcje OOP dostępne w PHP. Jednak przed rozpoczęciem lektury kolejnego rozdziału naprawdę warto utrwalić wiadomości przedstawione w bieżącym, aby uniknąć zakłopotania w przypadku niektórych zagadnień. Warto więc samodzielnie poćwiczyć i spróbować przenieść tworzony wcześniej kod proceduralny na styl OOP. Im więcej czasu czytelnik poświęci na praktykę, tym bardziej efektywnym programistą zostanie.